Move and Character Classification using Street Fighter

Jeff Lucca, Joseph Woolfolk, Shreyas Kadekodi
Team A

1 Introduction

In our project, we have attempted to create a classification system in order to classify characters and moves in the game Street Fighter. Street Fighter 3 is a competitive arcade game in which two players fight each other, featuring a wide variety of characters with different sets of moves and abilities. We are able to generate our data during a live match, and the intention is to allow the individual to respond better to the opponent using this data. For example, a more aggressive approach might become apparent after classification, and the individual can respond and adapt to this data. A certain move could have an easy counter that could be easier to use if the move and characters are apparent.

2 Why is this project a good or bad idea?

This project, we feel, is a good idea. The project allows us to take something we enjoy - video games- and apply our knowledge to better enhance our skills. It has implications beyond that as well - the fundamentals of this project could also be used to build an AI fighter, or used on real people or objects, and so on. We have structured this project in a way that matches what we learnt in class. For instance, we chose to use a CNN, which is ideal for image classification, which this project is built around. And while Street Fighter itself may not have obvious applications to real life, being able to classify individuals and actions in real time is a very relevant topic in the current climate, with applications from autonomous cars to security.

3 What did we do?

Our project accomplishes two separate tasks: character tracking and move prediction. As such, we tackled each of those problems individually before combining them together for use on real-world game/video input. Both of these tasks utilize a custom dataset which we generated ourselves using image files taken from the game.

3.1 Initial Data Collection

No dataset is available for Street Fighter, so this left us with two options: manually labeling data ourselves or devising a way to generate it. Due to the many different variables involved in both character tracking and move classification - the character's player-chosen color or the stage color, to name a few - it would be non-trivial to manually label enough data that allows for suitable performance.

Instead, we use a Selenium-based script to download images extracted from the game from an archival website, `https://www.justnopoint.com/zweifuss/index.htm`. This website contains GIFs of every move animation performed by each character. With these images, we can take any version of any character that we want, make it perform some move or action, and overlay that onto the stage to create an image that looks identical to the real game apart from the UI overlay. The process by which we generate data differs for both of our models, but the core image files used to create that data is all downloaded with Selenium.

3.2 Character Tracking

To implement character tracking, we used both the Tensorflow object detection API and used SSD MobileNet v1 COCO. To generate data, we created a script to take the previously-downloaded GIFs, select a random color and a random move for a given character, split it into frames, and then crop out the transparency in each frame. Since some characters have moves that span a wide or tall area, the character can occupy one section of the frame while the rest is empty, so simply using the whole frame's height and width with no cropping wouldn't match the character's true bounding box. Then, these frames were then placed at a random coordinate over an image of a random stage in order to account for the backgrounds behind the characters. After placing the image, we also generated an accompanying XML file to document the character's bounding box that we calculate through the random coordinates we chose added to the height and width of the character we placed on it. This process was repeated 400 times for each character. 20% of the images were used for the test data, and 80% was training data. Before generating data using the Selenium images, we had initially manually tagged bounding boxes by using an emulator running Street Fighter and taking screenshots as gameplay progressed using a script to capture a window playing the game. We exported those screenshots to another program that allowed us to label and bound objects, called LabelImg. Manually marking was done by drawing a bounding box over the character in frames wherein the given character was demonstrating a unique move until every frame of the character's movement was captured.

With this training data, Tensorflow was then used to generate a bounding box for each character it was trained on. To achieve move classification, we must take the bounded character and send the image to the move classifier model. For this, we took the calculated bounding box, cropped our input image to find the character, and then passed the frames as input to the classifier model.

Figure 1: Correctly labelled image

3.3 Move Classification

For the move classification, we need to find a way to take the character-cropped image from our previous model and classify the move on every single frame. Since our character tracking model knows the class of the character already, we individually train one classifier per character and select the classifier to predict with based on which character's frames were passed. Since the game runs at 60 FPS, this means we're outputting 60 move classes per second, one per frame. To start designing the model, our intuition is that if we include multiple frames, the model will have a higher likelihood of making a correct prediction. One problem with classifying an individual frame is that individual frames across different moves may look similar to one another. For example, if you consider the first frame of some given move, the character's arms and legs haven't had much time to move yet, so it will look very close to the default standing position and would be easy to mix up with the first frame of other moves. Classifying across the entire move sequence gives the model more context to work with, since it has more frames from the same move class to help discriminate if a single frame is ambiguous. Since we have a time-stepped sequence of frames, the core of our model is an LSTM.

For each frame of the game, we classify the move based on the six most recent frames passed in by the character tracking model.

In order to extract high-level features from the image, we decided to use a pretrained CNN. This should help the model disambiguate between different shapes and edges rather than simply feeding the raw pixels into our LSTM. For our CNN, we decided to use SqueezeNet. SqueezeNet has relatively good performance while having fewer parameters than other CNNs, which gave us more computational efficiency to hit real-time classification at 60 FPS. Additionally, since our classification task seems simple with only 69 output classes for the character Ryu, the additional power provided by other models likely shouldn't matter. We replaced the last layer of SqueezeNet with a 1024-class layer for us to fine-tune on, then ran the output through our LSTM with a 512 hidden size and 3 layers. Finally, we apply a linear layer to the output of the LSTM, and find the move's class by taking the argmax.

Since our move label is represented by text, we use indexing to convert it to an integer tensor. We convert our label to an integer index to feed our label into the model during training, and then we convert our index output by the model back into a label to find the move classification during prediction.

4 Results

Our results for character tracking were generally quite accurate, as can be seen in figure 3. In general, characters had an accuracy score generally above 80%. There were two exceptions - Oro had an accuracy of 58%, and Remy had an accuracy of 0%.

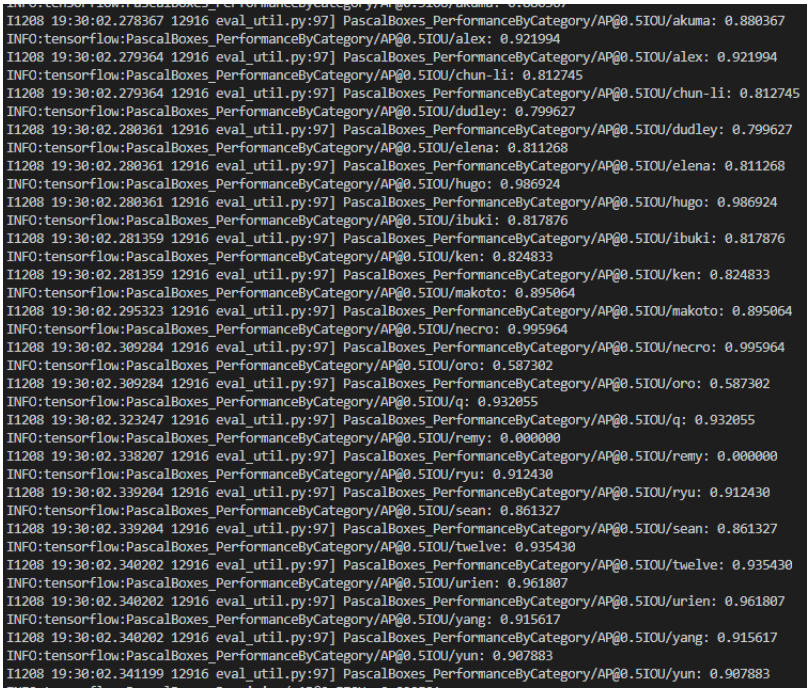

Figure 2: Detection accuracy for all characters

The issue with Remy wasn't due to the model - there were issues with the data generation, so we ended up skipping through his character. Urien was the most problematic character that we fully collected data for. Since Urien is naked, the model likely had hard time picking up on distinguishing features and isolating the character from the background.

As seen in Figure 3, the majority of Urien's body is similar to the background of this stage. When trying to perform character tracking, especially on this stage, the model would confuse the parts of the stage with Urien himself. The other major issue with character tracking was with Ibuki, who moves in a way unlike any other character with highly flashy moves that may make it difficult for the

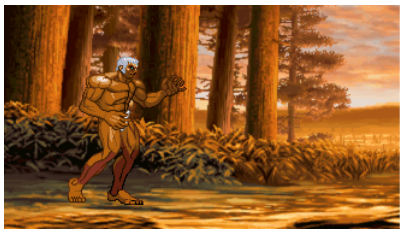

Figure 3: The Urien problem

model to find the distinguishing features of the model in a given frame. Other than these major issues the model would sometimes mislabel characters.

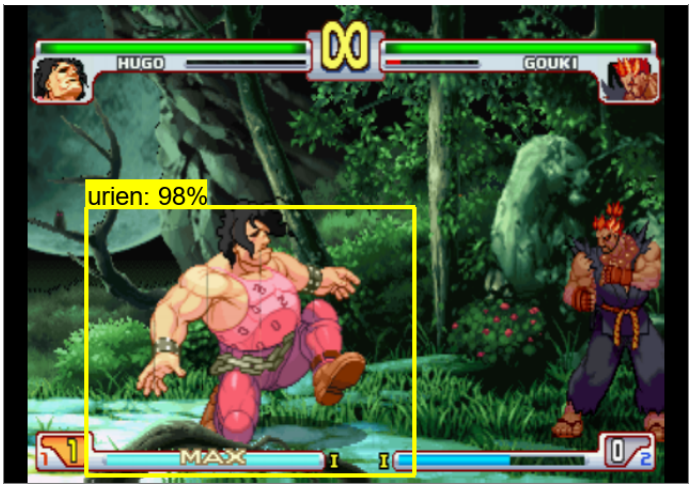

Figure 4: Mislabelled image

A way to alleviate this may be to increase the number images used in training, which was difficult to do during the project timeline because of hard drive space.

For the move classification, our results were poor, but still performed above the baseline of random guessing. We only trained one model for Ryu since the 50,000 training samples took up 30GB, but for Ryu, there are 69 different move classes. A random guess would lead to the correct classification about 1.45% of the time. Our model ended up with 5.7% training accuracy, which even while accounting for dataset imbalance, is more accurate than random guesses. Below, we list every move along with the number of times they have been correctly classified.

move index	correct	total
67	0	347
33	0	115
35	0	115
22	0	188
27	0	109
28	0	110
51	0	115
5	0	110
15	0	239
30	0	100
46	0	123
13	0	120
31	0	110
36	0	118
52	0	113
29	0	134
63	0	128
44	0	138
10	0	141
61	0	105
8	0	85
17	0	236
65	0	100
12	0	139
34	0	118
1	0	174
4	0	101
6	0	102
16	0	210
7	0	123
58	0	137
39	349	349
2	0	184
68	0	329
47	0	148
64	0	125
3	0	161
9	0	128
56	0	128
66	0	111
43	0	116
54	0	101
26	0	109
32	0	95
62	0	105
48	0	116
41	0	156
50	0	110
42	0	130
59	0	150
45	0	149

move index	correct	total
24	0	196
14	0	130
55	0	107
21	0	237
57	0	130
38	222	313
20	0	190
11	0	137
0	0	159
23	0	241
53	0	119
25	0	226
49	0	144
60	0	135
37	0	115
40	0	118

As we can see, the model is clearly overfitting to classes 39 and 38, which are Ryu's two super arts.

For class 38, we see something interesting. The GIF for this move features a transparently-flashing fireball, which caused the data generation to glitch and produce the fireball as the entire frame. From this, it's very simple for the model to identify the blue circle and fit to it.

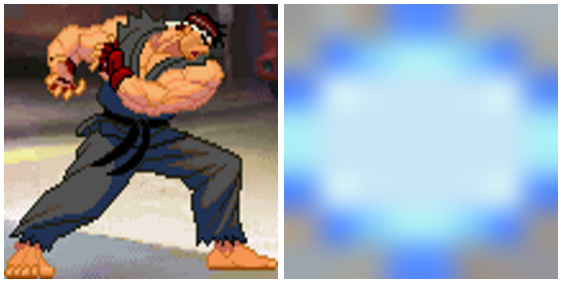

Figure 5: The first two frames of one sample for class 38.

Figure 6: The first four frames of one sample for class 39.

For class 39, however, the model likely chose it because there are more samples for it than for any other class. By predicting everything as 39 unless it has class 38's unique characteristics, the model achieves the best accuracy that it can trivially.

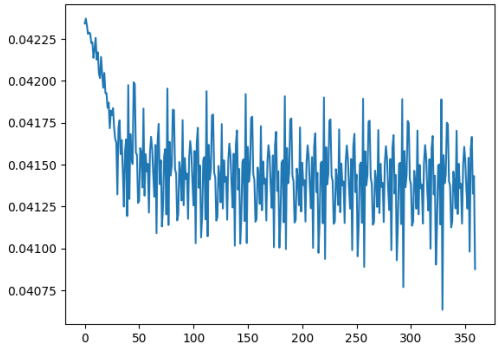

Figure 7: Training loss over 10 epochs (graph not normalized) with a batch size of 256.

Figure 7 contains the training loss for our model. While it decreases at first, it very quickly caps out as it starts to overfit to the two classes. This could be for a number of reasons. SqueezeNet's low parameter count doesn't seem to make a difference, as the model performs just as poorly using

DenseNet instead. The problem with learning may be that it's difficult for the CNN to extract features given that it was trained on ImageNet, which features real-life images, while our dataset features pixel art constructed with a fixed color palette. Given that the CNN+LSTM works for other time-stepped image sequence applications, there is likely to be a problem with the pretraining or the parameters/hyperparameters rather than the overall model architecture itself.

5 Who did what?

Jeff Lucca

Jeff largely worked on the move classification, the Selenium script to download move data, and the scripts to generate training and test data. He came up with the sequence of steps we would use for move classification, and implemented it. He choose squeezenet as the pre-trained CNN, and coded the LSTM. He also helped generate the data to allow for our error analysis. Along with the rest of the group, Jeff helped coordinate the sharing of data and code, as well as meetings.

Joseph Woolfolk

Joseph worked primarily on the character classification and integrating the move classification written by Jeff. He created the code to screen capture the emulator and bound the characters with a box, and ended up figuring out how to crop it. Joseph also helped manually bound certain characters for move classification. All testing with the actual game was done on Joseph's computer. Along with the rest of the group, Joseph helped coordinate the sharing of data and code, as well as meetings.

Shreyas Kadekodi

Shreyas' tasks consisted primarily of helping Joseph and Jeff with their respective sections, although this largely ended up being assistance with Joseph's section. While not his work ended up being used on the final project, Shreyas helped with the tensorflow and attempted to familiarize himself with YOLO in the early stages of the project. Shreyas helped with creating manual bounding boxes for specific characters as well, and created the outlines and slide for the presentation and wrote most of this paper. Along with the rest of the group, Shreyas helped coordinate the sharing of data and code, as well as meetings.